

08 | smart_ptr: 智能指针到底“智能”在哪里？

2020-05-23 罗剑锋

罗剑锋的C++实战笔记

[进入课程 >](#)



讲述: Chrono

时长 12:14 大小 11.21M



你好，我是 Chrono。

上节课在讲 `const` 的时候，说到 `const` 可以修饰指针，不过今天我要告诉你：请忘记这种用法，在现代 C++ 中，绝对不要再使用“裸指针 (naked pointer)”了，而是应该使用“智能指针 (smart pointer)”。

你肯定或多或少听说过、用过智能指针，也可能看过实现源码，那么，你心里有没有一种疑惑，智能指针到底“智能”在哪里？难道它就是解决一切问题的“灵丹妙药”吗？



学完了今天的这节课，我想你就会有明确的答案了。

什么是智能指针？

所谓的“智能指针”，当然是相对于“不智能指针”，也就是“裸指针”而言的。

所以，我们就先来看看裸指针，它有时候也被称为原始指针，或者直接简称为指针。

指针是源自 C 语言的概念，本质上是一个内存地址索引，代表了一小片内存区域（可能会很大），能够直接读写内存。

因为它完全映射了计算机硬件，所以操作效率高，是 C/C++ 高效的根源。当然，这也是引起无数麻烦的根源。访问无效数据、指针越界，或者内存分配后没有及时释放，就会导致运行错误、内存泄漏、资源丢失等一系列严重的问题。

其他的编程语言，比如 Java、Go 就没有这方面的顾虑，因为它们内置了一个“垃圾回收”机制，会检测不再使用的内存，自动释放资源，让程序员不必为此费心。

其实，C++ 里也是有垃圾回收的，不过不是 Java、Go 那种严格意义上的垃圾回收，而是广义上的垃圾回收，这就是**构造 / 析构函数**和 **RAII 惯用法**（Resource Acquisition Is Initialization）。

我们可以应用代理模式，把裸指针包装起来，在构造函数里初始化，在析构函数里释放。这样当对象失效销毁时，C++ 就会**自动**调用析构函数，完成内存释放、资源回收等清理工作。

和 Java、Go 相比，这算是一种“微型”的垃圾回收机制，而且回收的时机完全“自主可控”，非常灵活。当然也有一点代价——你必须要针对每一个资源手写包装代码，又累又麻烦。


智能指针就是代替你来干这些“脏活累活”的。它完全实践了 RAII，包装了裸指针，而且因为重载了 * 和 -> 操作符，用起来和原始指针一模一样。

不仅如此，它还综合考虑了很多现实的应用场景，能够自动适应各种复杂的情况，防止误用指针导致的隐患，非常“聪明”，所以被称为“智能指针”。

常用的有两种智能指针，分别是 **unique_ptr** 和 **shared_ptr**，下面我就来分别介绍一下。

认识 unique_ptr


unique_ptr 是最简单、最容易使用的一个智能指针，在声明的时候必须用模板参数指定类型：

 复制代码

```
1 unique_ptr<int> ptr1(new int(10));           // int智能指针
2 assert(*ptr1 == 10);                         // 可以使用*取内容
3 assert(ptr1 != nullptr);                     // 可以判断是否为空指针
4
5 unique_ptr<string> ptr2(new string("hello")); // string智能指针
6 assert(*ptr2 == "hello");                    // 可以使用*取内容
7 assert(ptr2->size() == 5);                   // 可以使用->调用成员函数
```


你需要注意的是，unique_ptr 虽然名字叫指针，用起来也很像，但它**实际上并不是指针，而是一个对象**。所以，不要企图对它调用 delete，它会自动管理初始化时的指针，在离开作用域时析构释放内存。

另外，它也没有定义加减运算，不能随意移动指针地址，这就完全避免了指针越界等危险操作，可以让代码更安全：

 复制代码

```
1 ptr1++;                                     // 导致编译错误
2 ptr2 += 2;                                 // 导致编译错误
```

除了调用 delete、加减运算，初学智能指针还有一个容易犯的错误是把它当成普通对象来用，不初始化，而是声明后直接使用：

 复制代码

```
1 unique_ptr<int> ptr3;                       // 未初始化智能指针
2 *ptr3 = 42;                                 // 错误！操作了空指针
```

未初始化的 unique_ptr 表示空指针，这样就相当于直接操作了空指针，运行时就会产生致命的错误（比如 core dump）。


为了避免这种低级错误，你可以调用工厂函数 `make_unique()`，强制创建智能指针的时候必须初始化。同时还可以利用自动类型推导（[第 6 讲](#)）的 `auto`，少写一些代码：

```
1 auto ptr3 = make_unique<int>(42);           // 工厂函数创建智能指针
2 assert(ptr3 && *ptr3 == 42);
3
4 auto ptr4 = make_unique<string>("god of war"); // 工厂函数创建智能指针
5 assert(!ptr4->empty());
```

 复制代码

不过，`make_unique()` 要求 C++14，好在它的原理比较简单。如果你使用的是 C++11，也可以自己实现一个简化版的 `make_unique()`，可以参考下面的代码：

```
1 template<class T, class... Args>           // 可变参数模板
2 std::unique_ptr<T>                         // 返回智能指针
3 my_make_unique(Args&&... args)             // 可变参数模板的入口参数
4 {
5     return std::unique_ptr<T>(              // 构造智能指针
6         new T(std::forward<Args>(args)...)); // 完美转发
7 }
```

 复制代码

unique_ptr 的所有权

使用 `unique_ptr` 的时候还要特别注意指针的“所有权”问题。

正如它的名字，表示指针的所有权是“唯一”的，不允许共享，任何时候只能有一个“人”持有它。

为了实现这个目的，`unique_ptr` 应用了 C++ 的“转移”（move）语义，同时禁止了拷贝赋值，所以，在向另一个 `unique_ptr` 赋值的时候，要特别留意，必须用 `std::move()` 函数显式地声明所有权转移。

赋值操作之后，指针的所有权就被转走了，原来的 `unique_ptr` 变成了空指针，新的 `unique_ptr` 接替了管理权，保证所有权的唯一性：

 复制代码

```

1 auto ptr1 = make_unique<int>(42); // 工厂函数创建智能指针
2 assert(ptr1 && *ptr1 == 42); // 此时智能指针有效
3
4 auto ptr2 = std::move(ptr1); // 使用move()转移所有权
5

```

如果你对右值、转移这些概念不是太理解，也没关系，它们用起来也的确比较“微妙”，这里你只要记住，**尽量不要对 unique_ptr 执行赋值操作**就好了，让它“自生自灭”，完全自动化管理。

认识 shared_ptr

接下来要说的是 shared_ptr，它是一个比 unique_ptr 更“智能”的智能指针。

初看上去 shared_ptr 和 unique_ptr 差不多，也可以使用工厂函数来创建，也重载了 * 和 -> 操作符，用法几乎一样——只是名字不同，看看下面的代码吧：

 复制代码

```

1 shared_ptr<int> ptr1(new int(10)); // int智能指针
2 assert(*ptr1 == 10); // 可以使用*取内容
3
4 shared_ptr<string> ptr2(new string("hello")); // string智能指针
5 assert(*ptr2 == "hello"); // 可以使用*取内容
6
7 auto ptr3 = make_shared<int>(42); // 工厂函数创建智能指针
8 assert(ptr3 && *ptr3 == 42); // 可以判断是否为空指针
9
10 auto ptr4 = make_shared<string>("zelda"); // 工厂函数创建智能指针
11 assert(!ptr4->empty()); // 可以使用->调用成员函数

```

但 shared_ptr 的名字明显表示了它与 unique_ptr 的最大不同点：**它的所有权是可以被安全共享的**，也就是说支持拷贝赋值，允许被多个“人”同时持有，就像原始指针一样。

 复制代码

```

1 auto ptr1 = make_shared<int>(42); // 工厂函数创建智能指针
2 assert(ptr1 && ptr1.unique() ); // 此时智能指针有效且唯一
3
4 auto ptr2 = ptr1; // 直接拷贝赋值，不需要使用move()
5 assert(ptr1 && ptr2); // 此时两个智能指针均有效
6
7 assert(ptr1 == ptr2); // shared_ptr可以直接比较

```

```
8 // 两个智能指针均不唯一，且引用计数为2
9 assert(!ptr1.unique() && ptr1.use_count() == 2);
10 assert(!ptr2.unique() && ptr2.use_count() == 2);
11
```

`shared_ptr` 支持安全共享的秘密在于**内部使用了“引用计数”**。

引用计数最开始的时候是 1，表示只有一个持有者。如果发生拷贝赋值——也就是共享的时候，引用计数就增加，而发生析构销毁的时候，引用计数就减少。只有当引用计数减少到 0，也就是说，没有任何人使用这个指针的时候，它才会真正调用 `delete` 释放内存。

因为 `shared_ptr` 具有完整的“值语义”（即可以拷贝赋值），所以，**它可以在任何场合替代原始指针，而不用再担心资源回收的问题**，比如用于容器存储指针、用于函数安全返回动态创建的对象，等等。

`shared_ptr` 的注意事项

那么，既然 `shared_ptr` 这么好，是不是就可以只用它而不再考虑 `unique_ptr` 了呢？

答案当然是否定的，不然也就没有必要设计出来多种不同的智能指针了。

虽然 `shared_ptr` 非常“智能”，但天下没有免费的午餐，它也是有代价的，**引用计数的存储和管理都是成本**，这方面是 `shared_ptr` 不如 `unique_ptr` 的地方。


如果不考虑应用场合，过度使用 `shared_ptr` 就会降低运行效率。不过，你也不需要太担心，`shared_ptr` 内部有很好的优化，在非极端情况下，它的开销都很小。

另外一个要注意的地方是 **`shared_ptr` 的销毁动作**。

因为我们把指针交给了 `shared_ptr` 去自动管理，但在运行阶段，引用计数的变动是很复杂的，很难知道它真正释放资源的时机，无法像 Java、Go 那样明确掌控、调整垃圾回收机制。

你要特别小心对象的析构函数，不要有非常复杂、严重阻塞的操作。一旦 `shared_ptr` 在某个不确定时间点析构释放资源，就会阻塞整个进程或者线程，“整个世界都会静止不


动”（也许用过 Go 的同学会深有体会）。这也是我以前遇到的实际案例，排查起来费了很多功夫，真的是“血泪教训”。

 复制代码

```
1 class DemoShared final          // 危险的类，不定时的地雷
2 {
3 public:
4     DemoShared() = default;
5     ~DemoShared()                // 复杂的操作会导致shared_ptr析构时世界静止
6     {
7         // Stop The World ...
8     }
9 };
10
```

shared_ptr 的引用计数也导致了一个新的问题，就是“**循环引用**”，这在把 shared_ptr 作为类成员的时候最容易出现，典型的例子就是**链表节点**。

下面的代码演示了一个简化的场景：

 复制代码

```
1 class Node final
2 {
3 public:
4     using this_type      = Node;
5     using shared_type     = std::shared_ptr<this_type>;
6 public:
7     shared_type          next;        // 使用智能指针来指向下一个节点
8 };
9
10 auto n1 = make_shared<Node>();      // 工厂函数创建智能指针
11 auto n2 = make_shared<Node>();      // 工厂函数创建智能指针
12
13 assert(n1.use_count() == 1);        // 引用计数为1
14 assert(n2.use_count() == 1);
15
16 n1->next = n2;                      // 两个节点互指，形成了循环引用
17 n2->next = n1;
18
19 assert(n1.use_count() == 2);        // 引用计数为2
20 assert(n2.use_count() == 2);        // 无法减到0，无法销毁，导致内存泄漏
```

在这里，两个节点指针刚创建时，引用计数是 1，但指针互指（即拷贝赋值）之后，引用计数都变成了 2。

这个时候，`shared_ptr` 就“犯傻”了，意识不到这是一个循环引用，多算了一次计数，后果就是引用计数无法减到 0，无法调用析构函数执行 `delete`，最终导致内存泄漏。

这个例子很简单，你一下子就能看出存在循环引用。但在实际开发中，指针的关系可不像例子那么清晰，很有可能会不知不觉形成一个链条很长的循环引用，复杂到你根本无法识别，想要找出来基本上是不可能的。

想要从根本上杜绝循环引用，光靠 `shared_ptr` 是不行了，必须要用到它的“小帮手”：**`weak_ptr`**。

`weak_ptr` 顾名思义，功能很“弱”。它专门为打破循环引用而设计，只观察指针，不会增加引用计数（弱引用），但在需要的时候，可以调用成员函数 `lock()`，获取 `shared_ptr`（强引用）。

刚才的例子中，只要你改用 `weak_ptr`，循环引用的烦恼就会烟消云散：

 复制代码

```
1  class Node final
2  {
3  public:
4      using this_type      = Node;
5
6      // 注意这里，别名改用weak_ptr
7      using shared_type    = std::weak_ptr<this_type>;
8  public:
9      shared_type          next;    // 因为用了别名，所以代码不需要改动
10 };
11
12 auto n1 = make_shared<Node>(); // 工厂函数创建智能指针
13 auto n2 = make_shared<Node>(); // 工厂函数创建智能指针
14
15 n1->next = n2;                // 两个节点互指，形成了循环引用
16 n2->next = n1;
17
18 assert(n1.use_count() == 1);  // 因为使用了weak_ptr，引用计数为1
19 assert(n2.use_count() == 1);  // 打破循环引用，不会导致内存泄漏
20
21 if (!n1->next.expired()) {    // 检查指针是否有效
22     auto ptr = n1->next.lock(); // lock()获取shared_ptr
```



```
23     assert(ptr == n2);  
24 }
```

小结

好了，今天就先到这里。智能指针的话题很大，但是学习的时候我们不可能一下子把所有知识点都穷尽，而是要有优先级。所以我会捡最要紧的先介绍给你，剩下的接口函数等细节，还是需要你根据自己的情况，再去参考一些其他资料深入学习的。

我们来回顾一下这节课的重点。

1. 智能指针是代理模式的具体应用，它使用 RAII 技术代理了裸指针，能够自动释放内存，无需程序员干预，所以被称为“智能指针”。
2. 如果指针是“独占”使用，就应该选择 `unique_ptr`，它为裸指针添加了很多限制，更加安全。
3. 如果指针是“共享”使用，就应该选择 `shared_ptr`，它的功能非常完善，用法几乎与原始指针一样。
4. 应当使用工厂函数 `make_unique()`、`make_shared()` 来创建智能指针，强制初始化，而且还能使用 `auto` 来简化声明。
5. `shared_ptr` 有少量的管理成本，也会引发一些难以排查的错误，所以不要过度使用。

我还有一个很重要的建议：

既然你已经理解了智能指针，就尽量不要再使用裸指针、`new` 和 `delete` 来操作内存了。

如果严格遵守这条建议，用好 `unique_ptr`、`shared_ptr`，那么，你的程序就不可能出现内存泄漏，你也就不需要去费心研究、使用 `valgrind` 等内存调试工具了，生活也会更“美好”一点。

课下作业

最后是课下作业时间，给你留两个思考题：

1. 你觉得 `unique_ptr` 和 `shared_ptr` 的区别有哪些？列举一下。

2. 你觉得应该如何在程序里“消灭” new 和 delete?

欢迎你在留言区写下你的思考和答案，如果觉得今天的内容对你有所帮助，也欢迎分享给你的朋友，我们下节课见。

课外小贴士

1. 最早的智能指针是C++98里的`auto_ptr`，但因为有一些缺陷，现在已经被废弃了，不建议使用。
2. 工厂函数`make_unique()`、`make_shared()`不是只返回智能指针对象那么简单，它内部也有优化，通常要比手写类型构造的效率更高。
3. `shared_ptr`还有很多高级用法，比如定制删除函数，不只是用`delete`释放内存，而是能够执行任意的代码，在第17讲会有一个简单的示例。
4. 有的资料不建议在函数的入口参数里使用`shared_ptr`，原因是成本高。我的意见是程序的正确性和安全性是第一位的，先放手去用，保证功能正确之后才是性能优化。

5. 除了课程里所说的三种智能指针，还存在于其他形式的智能指针，例如“侵入式智能指针”，可以把已存在引用计数的类“改造”成智能指针。

课程预告

6月-7月课表抢先看

充 ¥500 得 ¥580

赠「¥ 118 月球主题 AR 笔记本」



[【点击】图片, 立即查看 >>>](#)

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 07 | `const/volatile/mutable`: 常量/变量究竟是怎么回事?

下一篇 09 | `exception`: 怎样才能用好异常?

精选留言 (21)

 写留言



Eglinux

2020-05-23

C++ 这么小众吗？讲得这么好，如果写成书，完全可以看成另外一本 effective C++ 了，为什么才 3000 多订阅人数？

展开 ∨

作者回复: 已经不少了吧，相识就是缘分，笑。



6



Zivon

2020-05-26

罗老师，今天尝试使用智能指针改写双向链表的时候感觉实现很麻烦啊，请问在实现这种基本数据结构的时候需要使用智能指针吗

展开 ∨

作者回复: 基本的数据结构强调效率，用智能指针就有点成本略高，当然作为练手还是可以的。

智能指针最适合的应用场景是“自动资源管理”，链表还是不太合适，而且使用shared_ptr容易出现循环引用，改成weak_ptr会好一些。



2



有学识的兔子

2020-05-25

1. 我的理解是Unique pointer 是对象而不是指针，但重载了*和箭头，离开了对象的作用域就会被析构，所管理的资源在析构执行过程中被释放；那move操作是不是对对象进行拷贝了才得以传递？

shared_ptr 貌似也是对象，重点在于引用计数，对赋值拷贝计数+1，对于执行析构时计数-1，同时判断计数是否0，被管理的资源是否需要释放；...

展开 ∨

作者回复:

1.说的挺好。move操作是转移，不是拷贝，把原unique_ptr给“偷”到了另一个对象里，所有权也就同时转移了。

2.对，shared_ptr的关键就是引用计数，原来的名字叫counted_ptr。

3.weak_ptr是“弱引用”，只是观察目标，不增加计数，所以不会导致循环引用，可以再看标准文档。



silverhawk

2020-05-31

说实话，unique_ptr比起裸指针革命性进步，但是shared_ptr 有点over engineering，感觉有些时候用起来会很坑。不知道你怎么看

展开 ▾



Luke

2020-05-30

使用智能指针可以自动析构“资源”，隐含了指针管理的细节，从而提高了代码的安全性和易用性，但是这是否同时意味着效率下降？在极致追求执行速度的系统中，是否需要避免使用智能指针，依赖程序员自己管理裸指针的new和delete呢？

展开 ▾

作者回复: 追求极致性能，那当然还是要自己管理好了，但这样也就要自己承担安全的责任了。

建议用unique_ptr，它的速度与裸指针几乎相同，没有引用计数的成本。



java2c++

2020-05-29

老师文稿中的这段代码我改了下如果封装到一个函数里，可以被多线程并发调用吗？unique_ptr尽管有move可以进行转移，但是同一时刻ptr1是不是只允许转移到一个ptr2，而之前转移成功的后续逻辑还没有执行完

...

展开 ▾

作者回复: unique_ptr不是线程安全的，不要在多线程里用。

应该用shared_ptr，但它也只有最基本的线程安全保证，不能完全依赖它，具体要看文档里的精确描述。

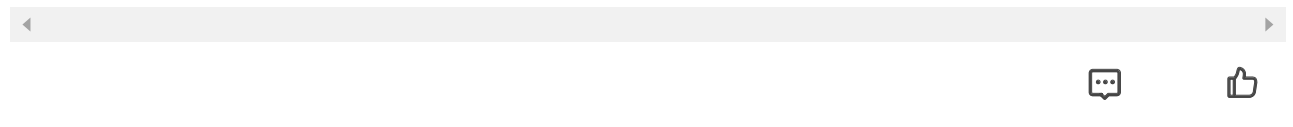


青生先森

2020-05-27

智能指针管理数组，是不是不会移动释放，需要自己手动书写？

作者回复: 不建议用智能指针管理数组，虽然这样也可以，最好用容器。

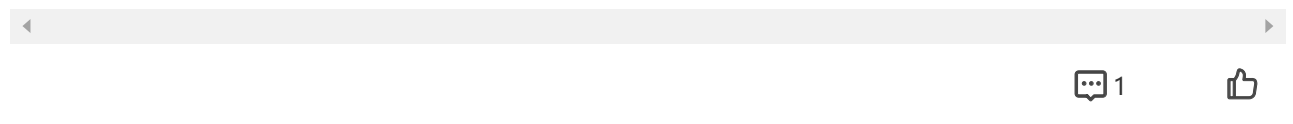


李锐

2020-05-26

罗老师，您好，我的工作场景中经常需要new一个数组来缓存从下位机采集到的数据，比如new [100]来缓存100帧图像数据，请问下，c++11中智能指针如何去管理new数组，谢谢

作者回复: 动态数组虽然也可以用shared_ptr来管理，但不是很推荐，其实更应该用容器vector，这个在C++98里就有。



Zivon

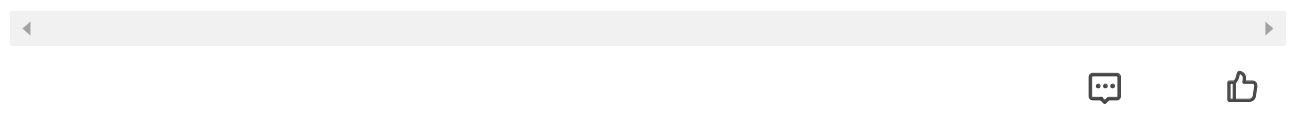
2020-05-25

shared_ptr 与unique_ptr最大区别就在于前者可以多个指针共享一个对象或元素，而后者一块内存空间只能由唯一的一个指针持有。

使用堆内存新建对象均使用智能指针，就可以不用new/delete

展开 ∨

作者回复: 说的很好，智能指针就要这样用。



禾桃

2020-05-25

“因为 shared_ptr 具有完整的“值语义”（即可以拷贝赋值）”

一直都觉得值语义这三个字比较难理解。想请教下这个概念到底是想说明什么问题，这个“值”该怎么理解？

展开 ∨

作者回复: 值是和引用对应的，值就是有实体，可以拷贝，而引用是虚的，只是个别名，操作上有区别。





lckfa李钊

2020-05-25

默认用的比较多的是unique_ptr，相反的shared_ptr倒是用的不多，因为担心文中提到的循环引用，资源消耗，线程安全等问题。大部分时候，unique_ptr是能完全取代裸指针的。如果是存粹的标准C++代码，使用智能指针确实很舒服，把它们当成一个普通的类型看就行了。但是，同时，作为C++程序员我们又不得不和裸指针打交道，不论是Linux还是Windows，我们不可避免的要使用它们的系统api，于是就不得不使用get将智能指针转...
展开 ∨

作者回复: 是的，涉及到系统底层，有时候就可能要用到delete，不过你也可以试着用RAII来管理，或者用shared_ptr的定制删除函数，还是能够找到不用delete的方式的。



fl260919784

2020-05-24

罗老师好，咨询个shared_ptr与多态的问题：

```
class A {public: virtual ~A() = default;}; //虚析构  
class B: public A{}; //共有继承
```

```
class HolderA {public: std::shared_ptr<A> data;}; //最少知道原则，只需持有A即可...  
展开 ∨
```

作者回复: 可以用std::dynamic_pointer_cast，在shared_ptr里存储基类指针，然后再动态转型成子类。

shared_ptr很灵活，有很多形式的构造函数和辅助工具，你说的这种情形很常见，所以肯定考虑到了，需要看接口文档，找出合适自己的解决办法。



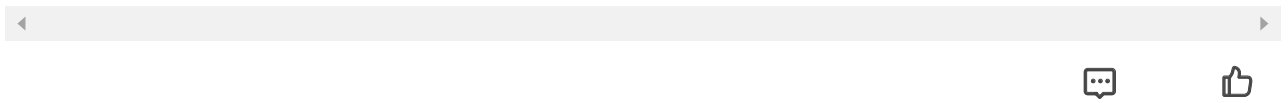
52rock

2020-05-24

开发的软件要在xp系统上运行貌似这些都不能用
展开 ∨

作者回复: xp系统太老了, 但新版的vc应该也支持编译出xp运行的应用吧。

抱歉很久没有做Windows开发了, 这方面不是太了解。

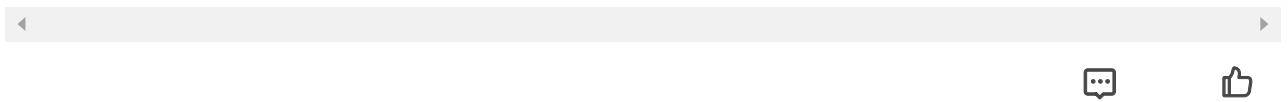


郭郭

2020-05-24

老师, 关于unique_ptr, 如auto ptr = ptr1, 那ptr1就应该被置空啦。不需要显示的调用std::move

作者回复: 我测试了一下, 是不行的, 会报编译错误, 因为unique_ptr禁止了普通的拷贝赋值, 只允许转移, 必须调用std::move()。



范闲

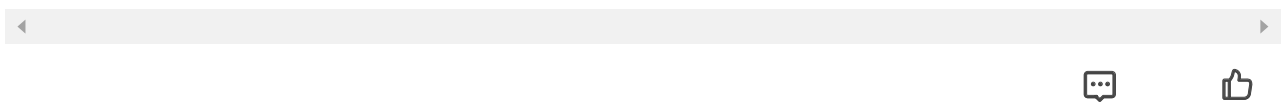
2020-05-23

1.unique的所有权只能转移, 不能增加。shared可以转移也可以增加。多个线程访问一个unique或shared也存在并发问题。unique没有循环引用的问题。

2. 尽量使用智能指针, 就可以避免手动管理了。

展开 ∨

作者回复: 说的很好, 有了智能指针, 就不需要new和delete了。



Confidant.

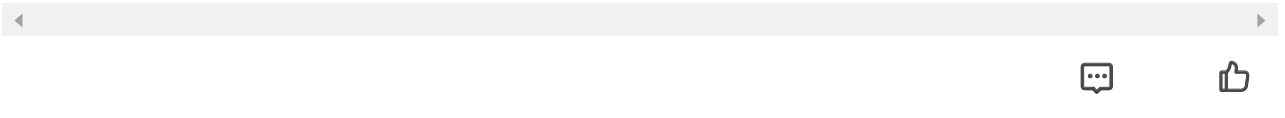
2020-05-23

问老师一个我最近被困惑到的智能指针相关的问题, 多线程的一些系统调用, 它需要传递一些指针来作为参数, 比如Linux或Windows底下的系统线程中子线程的参数, 比如CreateIoCompletionPort里面的参数, 我们从A线程传参, B线程取参, 中间可能经过了操作系统, 这种情况是无法使用智能指针的, 那是不是只能靠手动来管理呢?

展开 ∨

作者回复: shared_ptr也可以自定义删除函数, 自己定制管理策略, 这个就需要仔细分析资源的生存周期了, 涉及到底层必须小心。

C++提供的工具很多，用法也复杂，得仔细看接口文档，很可能就会在里面找到好的解决方案。

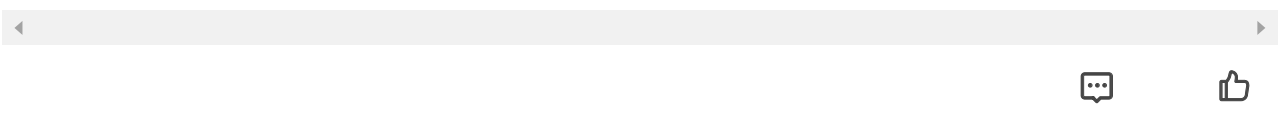


K. D.

2020-05-23

罗老师你好，有个问题请教一下：std::move是否也适用于shared_ptr，和=有区别吗？

作者回复: 区别很大，move是转移语义，=是拷贝共享语义，但对于shared_ptr来说一般没有必要用转移语义。



bearlu

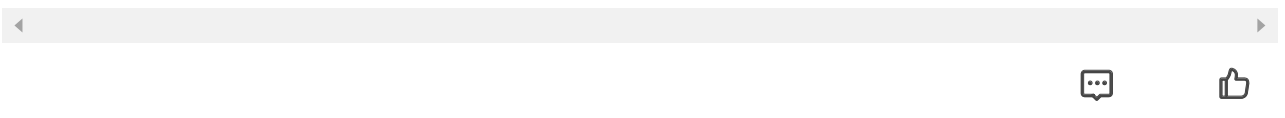
2020-05-23

shared_ptr 的销毁动作，这段说世界停止，是为什么停止，做了什么操作导致世界停止，智能指针不是线程安全？

展开 ∨

作者回复: 析构，里面有复杂操作阻塞线程。

shared_ptr只提供基本的线程安全，需要去细看文档，不能完全依赖它。



sea520

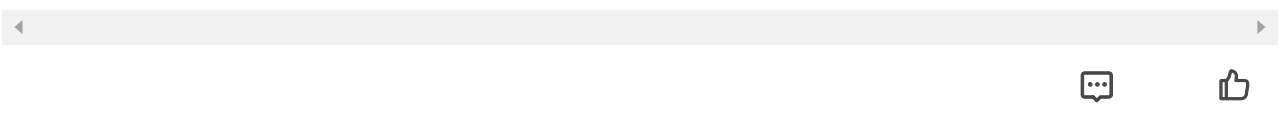
2020-05-23

您好！为什么面试喜欢问cpp底层实现细节。比如说虚表实现，stl实现？有什么最大的价值？万一工作换新语言还要都弄的很清楚吗？那时间成本呢？

展开 ∨

作者回复: 是啊，我觉得这样的面试官很low，可能他们的水平也就那样了吧。

不过既然是面试，也只能忍一下了，不过可以看出应聘公司的水平。

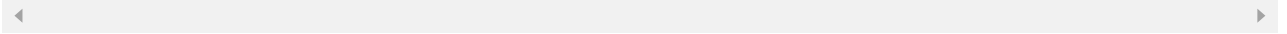




HoshinoKanade
2020-05-23

項目正在使用一個自家編寫的win32 api實現的引用計數器proxy，最近陷入了不可移植性的苦惱。請問老師有沒有辦法利用c++11的好處，而不需要停止使用這已經滲透在代碼庫所有角落的自家類別？

作者回复: boost里有一个intrusive_ptr，应该可以解决你的问题，可惜它不是标准库里的。



💬 2

